# Facilitating the Exploration of Compositions of Program Transformations

Albert Cohen, Sylvain Girbal, Olivier Temam

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Facilitating the Exploration of Compositions of Program Transformations*

Albert Cohen — Sylvain Girbal — Olivier Temam

**N° 5114**

Février 2004

THÈME 1


ISSN 0249-6399 ISRN INRIA/RR--5114--FR+ENG

INRIA FUTURS

# Facilitating the Exploration of Compositions of Program Transformations

Albert Cohen* , Sylvain Girbal*†‡ , Olivier Temam*†

Thème 1 —Réseaux et systèmes
Projet ALCHEMY

Rapport de recherche n° 5114 —Février 2004 —29 pages

**Abstract:** Static cost models have a hard time coping with hardware components exhibiting complex run-time behaviors, calling for alternative solutions. Iterative optimization is emerging as a promising research direction, but currently, it is mostly limited to finding the parameters of program transformations. We want to extend the scope and efficiency of iterative optimization techniques by searching not only for the appropriate parameters of a given transformation, but for the program transformations themselves, and especially for *compositions* of program transformations.

The purpose of this article is to introduce a framework for easily expressing compositions of program transformations. This framework relies on a unified polyhedral representation of loops and statements. The key is to clearly separate the impact of each program transformation on the following three components: the iteration domain, the schedule and the memory access functions. We show that, within this framework, composing a long sequence of program transformations induces no code size explosion. As a result, searching for compositions of transformations is not hampered by the multiplicity of compositions, and in many cases, it is equivalent to testing different values for the coefficients of the representation matrices. Our techniques have been implemented on top of the Open64/ORC compiler.

**Key-words:** iterative optimization, polytope model, loop-restructuring compiler.

* ALCHEMY Group, INRIA Futurs
† LRI, Paris South University
‡ CEA LIST, Saclay

Unité de recherche INRIA Futurs
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 —Télécopie : +33 1 39 63 53 30

# Facliliter l'exploration de compositions de transformations de programmes

**Résumé :** Les modèles de coût statiques s'accommodent très difficilement des composants matériels au comportement dynamique complexe, ce qui suggère des solutions alternatives. L'optimisation itérative apparaît comme une voie de recherche prometteuse, mais pour l'instant, elle se limite principalement a la recherche des paramètres des transformations de programme. Nous voulons étendre l'envergure et l'efficacité des techniques d'optimisation itérative en recherchant non seulement les paramètres apropriés d'une transformation donnée, mais aussi les transformations elles mêmes, et particulièrement les *compositions* de transformations de programme.

Cet article a pour but d'introduire un cadre général pour exprimer facilement des compositions de transformations de programme. Ce cadre général est fondé sur une représentation polyédrique unifiée des boucles et des instructions. L'idée directrice consiste a séparer radicalement l'impact de chaque transformation sur les trois composantes suivantes : le domaine d'itération, l'ordondancement, et les fonctions d'accès à la mémoire. Nous montrons que, dans ce cadre général de travail, composer des transformations selon une longue séquence n'induit pas d'explosion de la taille du code. Ainsi, la recherche de compositions de transformations n'est pas limitée par la multiplicité des compositions, et bien souvent, elle apparaît comme équivalente a l'essai de différentes valeurs pour les coefficients des matrices de la représentation. Nos techniques sont implémentées dans l'environnement du compilateur Open64/ORC.

**Mots-clés :** optimization itérative, modèle polyédrique, transformation de boucles à la compilation.

# 1 Introduction

Both high-performance and embedded architectures include an increasing number of hardware components with complex run-time behavior, e.g., cache hierarchies (including write buffers, TLBs, miss address files, L1 and L2 prefetching...), branch predictors, trace cache, load/store queue speculation, pipeline replays... Static cost models have a hard time coping with such hardware components, calling for alternative solutions. Iterative and feedback-directed optimizations are emerging as a promising research direction, with several recent research works [19, 24, 16] suggesting that optimizations based on dynamic information can better harness complex architectures and can be effectively used in practice [12].

Current approaches to iterative optimizations usually choose a rather small set of transformations, e.g., cache tiling, unrolling and array padding, and focus on finding the best possible transformation parameters, e.g., tile size, unroll factor and padding size [24] using parameter search space techniques. However, a recent comparative study of model-based versus empirical optimizations [36] outlines that many motivations for iterative optimization are irrelevant when the proper transformations are not available. Another recent study [26] also outlines that complex compositions of many distinct transformations can bring significant performance benefits.

We want to extend the scope and efficiency of iterative optimization techniques by searching not only for the appropriate parameters of a given program transformation, but for the program transformations themselves, and especially for *compositions* of program transformations; the sequence of program transformations then becomes a set of parameters in the search space, in addition to the program transformations parameters. For that reason, we need a generic method for expressing program transformations with the following properties: (1) allowing the easy composition of program transformations, (2) allowing compositions of program transformations to be searched in a systematic and practical way.

The ability to search for compositions of program transformations using existing compilers is limited. Compilers can embed a large array of transformations, but they are often expressed as a collection of adhoc syntactic transformations based on pattern-matching. In addition, control structures are regenerated after each program transformation, sometimes making it harder to apply the next transformations. Moreover, compilers follow a rigid ordering of phases, so that only short and fixed sequences of program transformations can be applied [34]. O'Boyle et al. [24] and Cooper et al. [16] have outlined that the ability to perform long sequences of composed transformations is key to the emergence of iterative optimization frameworks.

This article introduces a framework for easily expressing and searching compositions of program transformations. This framework relies on the polyhedral representation of loops and statements [11]. The key to our framework is to clearly separate the three dif-

ferent types of actions performed by program transformations: modification of the iteration domain (loop bounds and strides), modification of the schedule of the iteration of each individual statement, and modification of the memory access functions (array subscripts). For instance, loop interchange or loop fusion modify the schedule, strip mining and unrolling modify the iteration domain, and padding or privatization modify the memory access functions. However, current representations of program transformations do not clearly separate these three types of actions; as a result, the implementation of program transformations and especially the composition of those is complicated. For instance, current implementations of loop fusion must include loop bounds and array subscript modifications even though they are only byproducts of a schedule-oriented program transformation; after applying loop fusion, target loops are often peeled, increasing code size and making further optimizations more complex. Within our representation, loop fusion is only expressed as a schedule transformation, and the modifications of the iteration domain and memory access functions are implicitly handled. Similarly, a domain-oriented transformation like unrolling has no impact on the schedule or memory access functions representations; or a memory access function-oriented transformation like padding has no impact on the schedule or iteration domain representations, thus not conflicting with the later application of skewing or tiling.

Within that representation, each type of actions is represented in a parameterized way using three matrices per statement, so that any program transformation is represented as a set of operations on these matrices. The output of any transformation is a number of statements with a modified matrix triplet, and consequently, the output of any composition of transformations is again a number of statements with a modified matrix triplet. As a result, searching for compositions of transformations is not hampered by the multiplicity of compositions, and in many cases, it is equivalent to testing different values of the matrices parameters. Besides, with this framework, it should also be possible to find new compositions of transformations for which no static model has yet been developed, either because of their complex interplay with other transformations, their restricted scope or their possibly negative impact on performance, e.g., array expansion and speculation [9, 31, 4, 29].

Up to now, such compositions of transformations were only possible for unimodular transformations [34]. To date, the most thorough application of the polyhedral representation is the Petit dependence analyzer and loop restructuring tool [17] within the Omega project [18]. These tools show that most single loop transformations (both unimodular and non-unimodular) can be modeled as geometric transformations of polyhedra. However, this representation does not separate the three abovementioned actions induced by program transformations. As a result, it is not appropriate for complex compositions of transformations. Because space-time transformations in the polytope model [11, 17] were aimed at model-based optimizations through operations research algorithms, e.g., linear

programming, a black-box approach emerged with no real need for composition sequences. Although we could have built our new representation on top of Omega and benefit from the high level abstraction of Presburger arithmetic, we prefered the more focused PolyLib [21] for its robustness and for the scalability of its algorithms.[1] Similarly, we use a code generation technique suitable to a polyhedral representation that is again significantly more robust than the code generation proposed in the Omega library [1]. For instance, full benchmarks have been successfully handled and nests with more than 1700 statements could be generated with optimal control structures [2].

Note also that previous research works suggest that major potential performance benefits rest with across nests optimizations [22], and that simultaneously considering all loop nests may be a better iterative optimization strategy [24]. By considering each statement individually, this framework facilitates the development of complex across loop nest transformations such as forward substitution and copy propagation. Using precise array data-flow analyses [9, 5, 35], it is possible to generate compensation code across loop nests automatically, e.g., copy-in/out after privatization. More generally, facilitating across loop nests optimization is one step towards the general goal of achieving whole program optimization on a given architecture using iterative optimization.

While our framework is geared toward iterative optimization techniques, it can also facilitate the implementation of statically driven program transformations in a traditional optimizing compiler. The framework operates at an abstract semantical level to hide the details of the control structures, rather than on a syntax tree. Consequently, it eases the extension of existing transformation techniques, allowing per-statement and versatile transformations that make few assumptions about control structures and loop bounds, e.g., allowing triangular loops and non-convex domains.

This article is organized as follows. Section 2 presents our program and transformation framework, Section 3 shows how this representation facilitates the composition of transformations and searching for such compositions, and Section 4 describes the implementation of our representation and the associated code generation technique in Open64/ORC [25].

## 2 Separate Polyhedra for Unified Program Transformations

Compared to syntax tree approaches to program transformations, the polytope model is based on a more specific *semantics*-based representation of loop nests, abstracting away many implementation details and relying on the underlying polyhedral tools [10, 21, 1] to avoiding pattern matching. This semantics-based representation clearly identifies three

[1] Based on the dual representation of polyhedra and Chernikova's algorithm, compared to Fourier-Motzkin in the Omega test.

separate components: *array access functions* —affine functions describing the mapping of iterations to memory locations —from the *iteration domain* —a geometrical abstraction of loop bounds and strides shaping loop structures —and from the *affine schedule* —another geometrical abstraction of the ordering of iterations and statements. In addition to classical characterization of affine schedules, we also separate the description of iteration ordering of a single statement from inter-statements scheduling. In the following, we will use the term *separation principle* to refer to these separation properties. Despite their expressive power and flexibility, affine schedules and the separation principle seem overlooked in optimizing compiler approaches.[2] Our approach takes the form of a *unified representation of program transformations* that separates the three abovementioned components.

For the sake of the explanation, we consider the very simple example of Figure 1: the three loops can be fused to improve temporal locality of accesses to arrays `A` and `B`, and, assuming `A` is a local array not used outside the code fragment, it can be replaced with a scalar `a`. Figure 2 shows the corresponding optimized code. Both figures also show a graphical representation of the separate domains, schedules and access functions for the three statements *A*, *B* and *C* of the original and optimized versions. Notice the middle loop in Figure 1 has a reduced domain. In this example, optimizations mainly consist in loop fusions, which only have a visible impact on scheduling. In addition, the domain of statement *A* is reduced since the last iteration (1000) is useless, and the access function to array `A` disappeared since the array was replaced with a scalar.

**Limitations of syntactic representations.** We try to optimize this example using modern loop-restructuring compilers, namely Intel Electron (IA64) and ORC (IA64).[3] These compilers classically rely on pattern-matching techniques to look for transformation opportunities. Those techniques are fairly fragile because previous transformations may break target patterns of further ones. To better understand the interplay of loop peeling, loop fusion, scalar promotion and dead-code elimination, we first assume that `A` is a global (escaping) array, effectively restricting ourselves to peeling and fusion.

The reduced domain of *B* has no impact on our framework, which succeeds in fusing the three loops and yields the code in Figure 3. However, to fuse those loops, syntactic transformation frameworks require some iterations of the first and third loop to be peeled and interleaved between the loops. ORC was able to peel the last iteration to only fuse the first two loops, Figure 4; Electron failed to cope with the unbalanced iteration counts.

Now, because the pattern for loop fusion in ORC only matches consecutive loops, peeling prevents fusion with the third loop, as shown in Figure 4; we checked that neither a

[2] This is not the case in automatic parallelization [11, 18, 20, 13].

[3] ORC can produce source code.

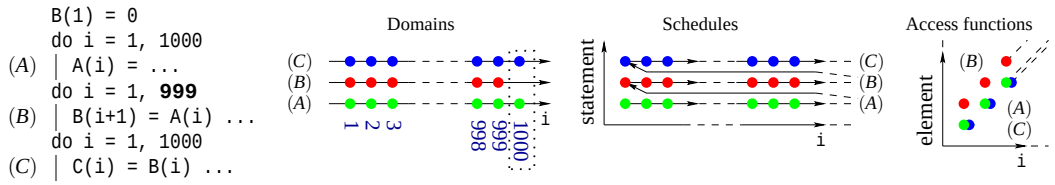


Figure 1: Original program and graphical view of its polyhedral representation

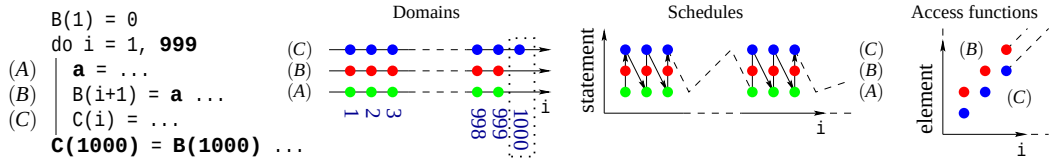


Figure 2: Target optimized program and graphical view

```
B(1) = 0
do i = 1, 999
| A(i) = ...
| B(i+1) = A(i) ...
| C(i) = B(i) ...
A(1000) = ...
C(1000) = B(1000) ...
```

Figure 3: Fusion of the three loops

```
B(1) = 0
do i = 1, 999
| A(i) = ...
| B(i+1) = A(i) ...
A(1000) = ...
do i = 1, 1000
| C(i) = B(i) ...
```

Figure 4: Peeled iteration prevents fusion

```
B(1) = 0
do i = 1, 999
| a = ...
| A(i) = a
| B(i+1) = a ...
do i = 1, 1000
| C(i) = B(i) ...
```

Figure 5: Dead code elimination first

```
B(1) = 0
do i = 1, 999
| a = ...
| B(i+1) = a ...
do i = 1, 1000
| C(i) = B(i) ...
```

Figure 6: Scalar promotion first

```
    B(1) = 0
    do i = 1, 1000
(A) | A(i) = A(1) ...
    do i = 1, 999
(B) | B(i+1) = A(i) ...
    do i = 1, 1000
(C) | C(i) = A(i) + B(i)
```

Figure 7: Advanced example

```
B(1) = 0
do i = 1, 999
| A(i) = A(1) ...
| B(i+1) = A(i) ...
| C(i) = B(i) ...
A(1000) = A(1) ...
C(1000) = A(1000) + B(1000)
```

Figure 8: Fusion of the three loops

```
B(1) = 0
do i = 1, 999
| A(i) = A(1) ...
| B(i+1) = A(i) ...
A(1000) = A(1) ...
do i = 1, 1000
| C(i) = A(i) + B(i)
```

Figure 9: Peeled dependences prevent fusion

failed dependence test nor an erroneous evaluation in the cost model may have caused the problem. Within our transformation framework, it is possible to fuse loops with different domains without prior peeling transformations because hoisting of control structures is delayed until code generation.

Against common belief, pattern matching is not the only limitation to compositions. Consider the example of Figure 7 which adds two references to the original program, `A(1)` in statement $(A)$ and `A(i)` in statement $(C)$. These references do not compromise the ability

to fuse the three loops, as shown in Figure 8. Optimizers based on more advanced rewriting systems [33] and most non-syntactic representations [23] will still peel an iteration of the first and last loops. However, peeling the last iteration of the first loop introduces two dependences that prevent fusion with the third loop: backward motion —flow dependence on `A(1)` —and forward motion —anti-dependence on `A(i)` —of the peeled iteration is now illegal. Electron cannot cope with this example and ORC yields the partially fused code in Figure 9, whereas our framework may still fuse the three loops as in Figure 8.

**Limitations of phase ordering.** To address the issue of transformation composition, compilers come with an ordered set of phases, each phase running some of the compiler optimizations or analyses. This phase ordering has a major drawback: it prevents transformations to be applied several times, after some other "enabling" transformation has modified the applicability or adequation of further optimizations. We consider again the example of Figure 1, and we now assume `A` is a local array only used to compute `B`. ORC applies dead-code elimination before fusion: it tries to eliminate `A`, but since it is used to compute `B`, it fails. Then the compiler fuses the two loops, and scalar promotion replaces `A` with a scalar, as shown in Figure 5. It is now obvious that array `A` can be eliminated but the dead-code elimination will not be run again. Conversely, if we delayed dead-code elimination until after loop fusion (and peeling), we would still not fuse with the third loop but we would eliminate `A` as well as the peeled iteration, as shown in Figure 6. Clearly, both phase orderings lead to sub-optimal results. However, if we compile the code from Figure 5 with ORC —as if we applied the ORC sequence of transformations twice —array `A` and the peeled iteration are eliminated, allowing the compiler to fuse the three loops, eventually reaching the target optimized program of Figure 2.

These simple examples illustrate the artificial restrictions to transformation composition and the consequences on permuting or repeating transformations in current syntactic compilers. Beyond parameter tuning, existing compilation infrastructures may not be very apropriate for iterative compilation. By design, it is hard to modify either phase ordering or transformation selection, and it is even harder to get any transformation pattern to match a significant part of the code after a long sequence of transformations. In addition, given the complex interplay of the applicability constraints and transformation side-effects, the structure of the search space is hard to understand.

## 2.1 Static Control Parts

Let us now define the principles of our unified polyhedral representation. The scope of our representation are loop nests amenable to an exact representation in the *polytope model* [27]. In particular, we assume constant strides and affine bounds; affine array substripts are hoped

for but not mandatory; and we do not consider interprocedural analyses and transformations, pointers and data-dependent control.

Loops are normalized and split in two categories: loops from 1 to some bound expression with an integer stride, called `do` loops; other kinds of loops are referred to as `while` loops. Within a function body, a *Static Control Part* (SCoP) is a maximal set of consecutive statements without `while` loops, where loop bounds and conditionals may only depend on invariants within this set of statements. These invariants include symbolic constants, formal function parameters and counters surrounding the SCoP: they are called the *global parameters* of the SCoP. This definition is a slight extension of *static control* nests [11]. We only consider SCoPs holding at least one loop, and the scope of a program transformation is always restricted to a given SCoP. An example with several SCoPs is shown in Figure 10, where the `while` loop does not belong to a SCoP.

```
integer x, y, A(N), B(N, 3*N+M-1), D(N), E(N)
do M = 1, 3
 ..........................................................................................
|  x = 0                                          |  SCoP 1, two statements
|  N = M*M                                        |    no loop: ignored
 ..........................................................................................
|  while (x .le. 100)
 ..........................................................................................
|  |  do i = 1, N                                 |  SCoP 2, five statements
|  |  |  A(i) = 0                          (S1)  |    parameters: M, N
|  |  |  do j=1, M                               |    iterators: k
|  |  |  |  A(i) = A(i) + B(i,2*i+j+N-1)   (S2)  |
|  |  D(1) = 1                             (S3)  |
|  |  do k = 3, N, 2                             |
|  |  |  D(k) = 2 * D(k-2)                 (S4)  |
|  |  |  E(k) = -A(k)                      (S5)  |
 ..........................................................................................
|  do p = 1, 9, 2                                 |  SCoP 3, two statements
|  |  x = x + A(p)                       (S6)  |    parameters: M
|  |  D(p) = D(p) - x                    (S7)  |    iterators: p
```

Figure 10: Decomposition into static control parts

Let us now introduce the formal representation of a SCoP and its elementary transformations. A static control part is a triple $(\mathcal{S}, \mathcal{A}, \mathbf{i}_{\text{gp}})$, where $\mathcal{S}$ is the set of consecutive *statements*, $\mathbf{i}_{\text{gp}}$ is the vector of *global parameters* of the SCoP, and $\mathcal{A}$ is the set of *abstract arrays* involved in the polyhedral representation (most arrays and some scalars). Vector $\mathbf{i}_{\text{gp}}$ is constant for the SCoP but statically unknown; yet its value is known at runtime, when entering the SCoP. $d_{\text{gp}} = \dim(\mathbf{i}_{\text{gp}})$ denotes the number of global parameters. Each abstract array $\mathcal{A}$ is a pair of an identifier and a *shape matrix* describing its dimensions.

For the second SCoP in Figure 10, $\mathcal{S} = \{S_1, \ldots, S_5\}$ and $\mathbf{i}_{\text{gp}} = \{N, M\}$, and

$$\mathcal{A} = \{(\mathtt{A}, \Sigma_{\mathtt{A}}), (\mathtt{B}, \Sigma_{\mathtt{B}}), (\mathtt{D}, \Sigma_{\mathtt{D}}), (\mathtt{E}, \Sigma_{\mathtt{E}}), (\mathtt{x}, \Sigma_{\mathtt{x}}), (\mathtt{y}, \Sigma_{\mathtt{y}})\};$$

shape matrices will be described in the next section.

## 2.2 Domains, Schedules and Access Functions

To implement the separation principle, we need to define the iteration domain, the statements schedule and the memory access functions of array references. We call $d^S$ the depth of statement $S$, i.e., the number of nested loops enclosing the statement in the SCoP. A statement $S \in \mathcal{S}$ is a quadruple $(\mathcal{D}^S, \mathcal{L}^S, \mathcal{R}^S, \theta^S)$, where $\mathcal{D}^S$ is the $d^S$-dimensional *iteration domain* of $S$, $\mathcal{L}^S$ and $\mathcal{R}^S$ denote array references written by $S$ (left-hand side) and read by $S$ (right-hand side) respectively, and $\theta^S$ is the *affine schedule* of $S$, defining the *sequential execution ordering* of iterations of $S$. We will use the second SCoP of Figure 10 to illustrate these definitions.

In the remainder of this text, the term *polyhedron* will be used in a broad sense to denote a *convex set of points in a lattice* (also called $\mathbb{Z}$-polyhedron or lattice-polyhedron), i.e., a set of points in a $\mathbb{Z}$ vector space bounded by affine inequalities. We will always denote matrices by capital letters, as opposed to vectors and functions.

**Iteration domains.** $\mathcal{D}^S$ is a *convex polyhedron* defined by matrix $\Lambda^S \in \mathcal{M}_{n, d^S + d^S_{\text{lv}} + d_{\text{gp}} + 1}(\mathbb{Z})$ such that

$$\mathbf{i} \in \mathcal{D}^S \iff \exists \mathbf{i}_{\text{lv}}, \Lambda^S \cdot \left(\mathbf{i}, \mathbf{i}_{\text{lv}}, \mathbf{i}_{\text{gp}}, 1\right)^t \geq 0$$

where $\Lambda^S$ is the matrix defining the domain inequalities; $n$ is the number of inequalities necessary to define the domain (the number of matrix rows, a priori not limited); $d_{\text{gp}}$ was defined in Section 2.1 as the number of SCoP global parameters; 1 adds a matrix column to specify the affine component of each domain inequality; and $d^S_{\text{lv}}$ is the number of local variables defined as follows. To represent arbitrary lattice polyhedra, a number $d^S_{\text{lv}}$ of *local variables* are added, when applicable, to each statement, in order to implement integer division and modulo operations via *affine projection*. Consider for instance statement $S_4$ of the second SCoP in Figure 10: a local variable $p$, defined by $\exists p, k = 3 + 2p$, is added to represent the odd values of $k$; consequently, $d^S_{\text{lv}} = 1$ (note also that $d^S = 1$ and $d_{\text{gp}} = 2$, see below).

Note that program statements guarded by non-convex conditionals —such as $1 \leq i \leq 3 \vee i \geq 8$ —are split into separate statements with convex domains in the polyhedral representation.

The domains of the five statements in the second SCoP are
$\mathcal{D}^{S_1} = \{i \mid 1 \leq i \leq N\}$,
$\mathcal{D}^{S_2} = \{(i,j) \mid 1 \leq i \leq N, 1 \leq j \leq M\}$,
$\mathcal{D}^{S_3} = \{()\}$ (the zero-dimensional vector),
$\mathcal{D}^{S_4} = \mathcal{D}^{S_5} = \{k \mid 3 \leq k \leq N \wedge \exists p, k = 3+2p\}$.
For instance, the $\Lambda$-matrices for statements $S_2$ and $S_4$ are

$$\Lambda^{S_2} = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & -1 & 0 & 1 & 0 \end{bmatrix} \quad \text{with} \quad \left| \begin{array}{l} \mathbf{i} = (i,j) \\ \mathbf{i}_{\text{lv}} = () \\ \mathbf{i}_{\text{gp}} = (N,M) \end{array} \right.$$

$$\Lambda^{S_4} = \begin{bmatrix} 1 & 0 & 0 & 0 & -3 \\ -1 & 0 & 1 & 0 & 0 \\ 1 & -2 & 0 & 0 & -3 \\ -1 & 2 & 0 & 0 & 3 \end{bmatrix} \quad \text{with} \quad \left| \begin{array}{l} \mathbf{i} = (k) \\ \mathbf{i}_{\text{lv}} = (p) \\ \mathbf{i}_{\text{gp}} = (N,M) \end{array} \right.$$

In $\Lambda^{S_2}$, the first two columns correspond to iterators $i, j$, the next two columns to the global parameters $N, M$ and the last column to the affine component; in $\Lambda^{S_4}$, the first column corresponds to iterator $k$, the second column to variable $p$, and the next columns as above ($N$, $M$, 1).

**Shape matrices and memory access functions.** For each abstract array $(\mathsf{A}, \Sigma_{\mathsf{A}})$ in $\mathcal{A}$, $\Sigma_{\mathsf{A}}$ characterizes an affine function from the global parameters to array dimensions. Formally, if $\mathsf{A}$ is a $d$-dimensional array, $\Sigma_{\mathsf{A}}$ is a $d_{\text{gp}}+1$ by $d$ matrix such that $\Sigma_{\mathsf{A}} \cdot (\mathbf{i}_{\text{gp}}, 1)$ is the vector of dimensions of the declaration of $\mathsf{A}$, following the Fortran conventions (column-major format, 1 is the first index), and assuming scalars are 0-dimensional arrays.

For instance,

$$\Sigma_{\mathsf{A}} = \Sigma_{\mathsf{D}} = \Sigma_{\mathsf{E}} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}, \Sigma_{\mathsf{B}} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 3 & -1 \end{bmatrix}, \Sigma_{\mathsf{x}} = \Sigma_{\mathsf{y}} = [].$$

$\mathcal{L}^S$ and $\mathcal{R}^S$ are *sets* of $(\mathsf{A}, f)$ pairs, where $\mathsf{A}$ is an array variable and $f$ is the *access function* mapping iterations in $\mathcal{D}^S$ to locations in $\mathsf{A}$. The access function $f$ is defined by a matrix $\mathrm{F} \in \mathcal{M}_{\dim(\mathsf{A}), d^S + d^S_{\text{lv}} + d_{\text{gp}} + 1}(\mathbb{Z})$ such that

$$f(\mathbf{i}) = \mathrm{F} \cdot \left(\mathbf{i}, \mathbf{i}_{\text{lv}}, \mathbf{i}_{\text{gp}}, 1\right)^t.$$

For instance, $\mathcal{L}^{S_2} = \{(\mathsf{A},(i))\}$ and $\mathcal{R}^{S_2} = \{(\mathsf{A},(i)),(\mathsf{B},(i,2*i+j-N-1)^t)\}$, are stored as

$$\mathcal{L}^{S_2} : \left\{\left(\mathsf{A},\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}\right)\right\}$$
$$\mathcal{R}^{S_2} : \left\{\left(\mathsf{A},\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}\right),\left(\mathsf{B},\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & -1 \end{bmatrix}\right)\right\}$$

with $\mathbf{i} = (i,j)$, $\mathbf{i}_{\text{lv}} = ()$ and $\mathbf{i}_{\text{gp}} = (N,M)$.

Since SCoPs may contain non-affine array subscripts, access functions do not describe all array references in general; non-affine ones must be handled conservatively in static analyses [5, 35, 29].

**Affine schedules.** $\theta^S$ is the *affine schedule* of $S$; it maps iterations in $\mathcal{D}^S$ to multidimensional *time stamps*, i.e., logical execution dates. Multidimensional time stamps are compared through the *lexicographic ordering* over vectors, denoted by $\ll$: iteration $\mathbf{i}$ of $S$ is executed before iteration $\mathbf{i}'$ of $S'$ if and only if $\theta^S(\mathbf{i}) \ll \theta^{S'}(\mathbf{i}')$.

$\theta^S$ is defined by a matrix $\Theta^S \in \mathcal{M}_{2d^S+1,d^S+d_{\text{gp}}+1}(\mathbb{Z})$ such that

$$\theta^S(\mathbf{i}) = \Theta^S \cdot (\mathbf{i},\mathbf{i}_{\text{gp}},1)^t.$$

Notice $\Theta^S$ does not involve local variables, since they would be redundant with the iterators they are related to. Notice also that the number of rows is $2d^S+1$ instead of $d^S$. To define the relative ordering of statements across iterations at depth $k$, we need $d^S$ dimensions (recall $d^S$ is the number of nested loops enclosing statement $S$ in the SCoP); we also need to define the relative ordering of statements within each iteration, i.e., we need an additional dimension for each depth plus depth 0, hence the $2d^S+1$. This encoding was first proposed by Feautrier [11] and used extensively by Kelly and Pugh [17].

The schedule matrix is decomposed in a form amenable to transformation composition and code generation; it consists of three sub-matrices: a square *iteration ordering matrix* $\mathrm{A}^S \in \mathcal{M}_{d^S,d^S}(\mathbb{Z})$ operating on iteration vectors, a *statement ordering vector* $\beta^S \in \mathbb{N}^{d^S+1}$, and $\Gamma^S \in \mathcal{M}_{d^S,d_{\text{gp}}+1}(\mathbb{Z})$ called a *parameterization matrix*. The structure of the schedule matrix $\Theta^S$ is shown below. The $\mathrm{A}^S_{i,j}$ coefficients capture the iteration order of statement $S$ with respect to its surrounding loop counters. The $\beta^S_i$ coefficients specify the ordering of $S$ among all other statements executed at the same iteration; the first row of $\Theta^S$ corresponds to depth 0, the outermost level.[4] The $\Gamma^S_{i,j}$ coefficients are added to extend the nature of possible transformations, allowing iteration advances and delays by constant or parametric amounts (an example using $\Gamma$ is presented in Section 3.2).

[4] Notice the first component of $\beta$ is numbered $\beta_0$.

$$\Theta^S = \begin{bmatrix} 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_0^S \\ A_{1,1}^S & \cdots & A_{1,d^S}^S & \Gamma_{1,1}^S & \cdots & \Gamma_{1,d_{gp}}^S & \Gamma_{1,d_{gp}+1}^S \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_1^S \\ A_{2,1}^S & \cdots & A_{2,d^S}^S & \Gamma_{2,1}^S & \cdots & \Gamma_{2,d_{gp}}^S & \Gamma_{2,d_{gp}+1}^S \\ \vdots & \ddots & \vdots & 0 & \ddots & 0 & \vdots \\ A_{d^S,1}^S & \cdots & A_{d^S,d^S}^S & \Gamma_{d^S,1}^S & \cdots & \Gamma_{d^S,d_{gp}}^S & \Gamma_{d^S,d_{gp}+1}^S \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_{d^S}^S \end{bmatrix} \quad (1)$$

Consider again statement $S_2$ in the second SCoP of the example in Figure 10, matrix $\Theta^{S_2}$ splits into

$$A^{S_2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \beta^{S_2} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \Gamma^{S_2} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Coefficients in A indicate that statement $S_2$ is executed every iteration $(i, j)$; the $\beta$ coefficients indicate the relative ordering of $S_2$ among all instructions executed at this iteration; $\Gamma$ can be null as in the above example, it is only used for transformations, see Section 3.2. Thus, the $\Theta$-matrix for $S_2$ is:

$$\Theta^{S_2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \text{ with } \left| \begin{array}{l} \mathbf{i} = (i, j) \\ \mathbf{i}_{lv} = () \\ \mathbf{i}_{gp} = (N, M) \end{array} \right.$$

and the corresponding schedule is $\theta^{S_2}(\mathbf{i}) = (0, i, 1, j, 0)$ The schedules of the other statements of this SCoP are: $\theta^{S_1}(\mathbf{i}) = (0, i, 0)^t$, $\theta^{S_2}(\mathbf{i}) = (0, i, 1, j, 0)$, $\theta^{S_3}(\mathbf{i}) = (1)$, $\theta^{S_4}(\mathbf{i}) = (2, k, 0)$, $\theta^{S_5}(\mathbf{i}) = (2, k, 1)$.

## 2.3 Enforcing Proper Program Representation

With the above framework, a given program can have multiple representations, and that, in turn, can limit the application of transformations. For instance, consider statements $B$ and $C$ in the first two loops of Figure 1; their $\beta$ vectors are respectively $\beta^B = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$ and $\beta^C = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$. Now, if we had used $\beta^C = \begin{bmatrix} 4 \\ 0 \end{bmatrix}$ instead, the schedule would be strictly identical

since the loop of statement $C$ is still scheduled after the loop of statement $B$, however there is now a gap between the depth-0 coefficients of statements $B$ and $C$. A condition for the application of fusion is that the target statements must be consecutive; otherwise, if there is a statement in between, either it can prevent fusion because of dependences, or at least, it is necessary to first decide where to move it. Consequently, in the above example, it is necessary to "normalize" the $\beta$ vector in order to forbid such gaps. Such normalization conditions are called *invariants*. Besides avoiding useless composition prohibitions, these invariants also serve to avoid matrix parameters overflow. The different invariants are listed below.

**Schedule density.** The purpose of this invariant is to ensure that all statements at an identical depth have a consecutive $\beta$ ordering (no gap). As a side-effect, this invariant also serves to avoid integer overflows on the $\beta$ parameters:

$$\beta_k^S > 0 \Rightarrow \exists S' \in \mathcal{S}, \mathrm{pfx}(\beta^S, k) = \mathrm{pfx}(\beta^{S'}, k) \wedge \beta_k^{S'} = \beta_k^S - 1, \tag{2}$$

where $\mathrm{pfx}(\beta^S, k)$ denotes the $k$ first dimensions of vector $\beta^S$; the condition states that, for any non-null $\beta$ parameter at dimension $k$, there necessarily exists another statement $S'$ with the same $k$-prefix and the preceding value at dimension $k$.

**Domain parameters.** The purpose of this invariant is to avoid redundant inequalities and integer overflows in the domain matrix $\Lambda^S$ parameters. For that purpose, we impose that the coefficients in a row of $\Lambda^S$ are always relatively prime:

$$\forall 1 \leq i \leq d^S, \gcd(\Lambda_{i,1}, \ldots, \Lambda_{i,d_{\mathrm{gp}}+1}) = 1. \tag{3}$$

Since these parameters are only used in the inequality $\Lambda^S \left(\mathbf{i}, \mathbf{i}_{\mathrm{lv}}, \mathbf{i}_{\mathrm{gp}}, 1\right)^t \geq 0$ (see previous section), factoring a parameter has no effect on the domain definition.

**Sequentiality.** The sequentiality invariant states that two distinct statements, or two identical statements in distinct iterations, cannot have the same time stamp:

$$S \neq S' \vee \mathbf{i} \neq \mathbf{i}' \Rightarrow \theta^S(\mathbf{i}) \neq \theta^{S'}(\mathbf{i}'). \tag{4}$$

A sufficient (though not necessary) condition to enforce that property is the following

$$|\det(\mathrm{A}^S)| = 1, \text{ i.e., } \mathrm{A}^S \text{ is unimodular, and } S \neq S' \Rightarrow \beta^S \neq \beta^{S'}. \tag{5}$$

## 2.4 Constructors

In our framework, program transformations will take the form of a set of elementary operations on the different matrices and vectors describing each statement in a SCoP. We first define the different elementary operations called *constructors*, and in the next section, we explain how to implement program transformations using these constructors.

Many matrix operations consist in adding or removing a row or column. Given a vector $v$ and matrix M with $\dim(v)$ columns and at least $i$ rows, $\mathsf{AddRow}(\mathrm{M}, i, v)$ inserts a new row at position $i$ in M and fills it with the value of vector $v$, whereas $\mathsf{RemRow}(\mathrm{M}, i)$ does the opposite transformation. Analogous constructors exist for columns, $\mathsf{AddCol}(\mathrm{M}, j, v)$ inserts a new column at position $j$ in M and fills it with vector $v$, whereas $\mathsf{RemCol}(\mathrm{M}, j)$ undoes the insertion. AddRow and RemRow are extended to operate on vectors.

Moving a statement $S$ is also a common operation. It only impacts the statement ordering vector $\beta^{S'}$ of some statements $S'$. Forward or backward movement of $S$ at depth $\ell$ triggers the same movement on every subsequent statement $S'$ at depth $\ell$ such that $\mathrm{pfx}(\beta^{S'}, \ell) = \mathrm{pfx}(\beta^{S}, \ell)$, where $\mathrm{pfx}(v, n)$ returns a length-$n$ prefix of $v$ (the vector built from its $n$ first components). Although rather intuitive, the following definition with prefixed blocks of statements is fairly technical. Consider a SCoP $\mathcal{S}$, a *statement ordering prefix* $P$ defining the depth at which statements should be moved, a *statement ordering prefix* $Q$ —prefixed by $P$ —marking the initial time-stamp of statements to be moved, and an offset $o$; $o$ is the value to be added/subtracted to the component at depth $\dim(P)$ of any statement ordering vector $\beta^S$ prefixed by $P$ and following $Q$. The move constructor $\mathsf{Move}(P, Q, o)$ leaves all statements unchanged except those satisfying the following conditions:

$$\forall S \in \mathcal{S}, P \sqsubseteq \beta^S \wedge (Q \ll \beta^S \vee Q \sqsubseteq \beta^S) : \beta^S_{\dim(P)} \leftarrow \beta^S_{\dim(P)} + o, \tag{6}$$

where $u \sqsubseteq w$ denotes that $u$ is a prefix of $v$. If $o$ is positive, $\mathsf{Move}(P, Q, o)$ inserts $o$ free slots before all statements $S$ sharing the same prefix $P$ and preceded by the statement ordering prefix $Q$, at the depth of $P$. Respectively, if $o$ is negative, $\mathsf{Move}(P, Q, o)$ deletes $-o$ slots. Notice these constructors make no assumption about the representation invariants and may violate them.

## 2.5 Primitives

From the earlier constructors, we will now define transformation *primitives* that enforce the invariants and serve as building blocks for higher level transformation sequences. Most primitives correspond to simple polyhedral operations. Figure 11 lists the main primitives affecting the polyhedral representation of a statement.[5] This figure uses the following nota-

[5] Many of these primitives can be extended to blocks of statements sharing a common statement ordering prefix (like the fusion and split primitives).

| Syntax | Prerequisites | Effect |
|---|---|---|
| UNIMODULAR$(S,\mathrm{U},\mathrm{V})$ | $U \in \mathcal{M}_{d^S,d^S}(\mathbb{Z})$ $\wedge \lvert\det(\mathrm{U})\rvert = \lvert\det(\mathrm{V})\rvert = 1$ | $\mathrm{A}^S \leftarrow \mathrm{U}.\mathrm{A}^S.\mathrm{V}$ |
| SHIFT$(S,\mathrm{M})$ | $\mathrm{M} \in \mathcal{M}_{d^S,d_{\mathrm{gp}}+1}(\mathbb{Z})$ | $\Gamma^S \leftarrow \Gamma^S + \mathrm{M}$ |
| PAD$(\mathsf{A},\mathrm{M})$ | $\mathrm{M} \in \mathcal{M}_{\dim(\mathsf{A}),d_{\mathrm{gp}}+1}(\mathbb{Z})$ | $\Sigma_{\mathsf{A}} \leftarrow \Sigma_{\mathsf{A}} + \mathrm{M}$ |
| CUTDOMAIN$(S,c)$ | $\dim(c) = d^S + d^S_{\mathrm{lv}} + d_{\mathrm{gp}} + 1$ | $\Lambda^S \leftarrow \mathsf{AddRow}\big(\Lambda^S, 0, c/\gcd(c_1,\ldots,c_{d^S+d^S_{\mathrm{lv}}+d_{\mathrm{gp}}+1})\big)$ |
| INSERT$(S,\ell)$ | $\ell \le d^S \wedge \beta^S_{\ell+1} = \cdots = \beta^S_{d^S} = 0$ $\wedge\big(\exists S' \in \mathcal{S}, \mathrm{pfx}(\beta^S,\ell+1) \sqsubseteq \beta^{S'}$ $\vee\, (\mathrm{pfx}(\beta^S,\ell),\beta^S_\ell - 1) \sqsubseteq \beta^{S'}\big)$ | $P = \mathrm{pfx}(\beta^S,\ell);\ \mathcal{S} \leftarrow \mathsf{Move}(P,(P,\beta^S_\ell),1) \cup S$ |
| DELETE$(S)$ | | $P = \mathrm{pfx}(\beta^S,d^S);\ \mathcal{S} \leftarrow \mathsf{Move}(P,(P,\beta^S_{d^S}),-1) \setminus S$ |
| EXTEND$(S,\ell)$ | | $d^S \leftarrow d^S + 1;\ \Lambda^S \leftarrow \mathsf{AddCol}(\Lambda^S,\ell,0);$ $\mathrm{A}^S \leftarrow \mathsf{AddRow}(\mathsf{AddCol}(\mathrm{A}^S,\ell,0),\ell,\mathbf{1}_\ell);$ $\beta^S \leftarrow \mathsf{AddRow}(\beta^S,\ell,0);\ \Gamma^S \leftarrow \mathsf{AddRow}(\Gamma^S,\ell,0);$ $\forall(\mathsf{A},\mathrm{F}) \in \mathcal{L}^S \cup \mathcal{R}^S, \mathrm{F} \leftarrow \mathsf{AddRow}(\mathrm{F},\ell,0)$ |
| ADDLOCALVAR$(S)$ | | $d^S_{\mathrm{lv}} \leftarrow d^S_{\mathrm{lv}} + 1;$ $\Lambda^S \leftarrow \mathsf{AddCol}(\Lambda^S, d^S+1, 0);$ $\forall(\mathsf{A},\mathrm{F}) \in \mathcal{L}^S \cup \mathcal{R}^S, \mathrm{F} \leftarrow \mathsf{AddCol}(\mathrm{F}, d^S+1, 0)$ |
| PRIVATIZE$(\mathsf{A},\ell,s)$ | | $\dim(\mathsf{A}) \leftarrow \dim(\mathsf{A}) + 1;\ \Sigma_{\mathsf{A}} \leftarrow \mathsf{AddRow}(\Sigma_{\mathsf{A}},\ell,s);$ $\forall S \in \mathcal{S}, \forall(\mathsf{A},\mathrm{F}) \in \mathcal{L}^S \cup \mathcal{R}^S;$ $\mathrm{F} \leftarrow \mathsf{AddRow}(\mathrm{F},\ell,\mathbf{1}_\ell)$ |
| CONTRACT$(\mathsf{A},\ell)$ | | $\dim(\mathsf{A}) \leftarrow \dim(\mathsf{A}) - 1;\ \Sigma_{\mathsf{A}} \leftarrow \mathsf{RemRow}(\Sigma_{\mathsf{A}},\ell);$ $\forall S \in \mathcal{S}, \forall(\mathsf{A},\mathrm{F}) \in \mathcal{L}^S \cup \mathcal{R}^S;$ $\mathrm{F} \leftarrow \mathsf{RemRow}(\mathrm{F},\ell)$ |
| FUSE$(P,o)$ | | $b = \max\{\beta^S_{\dim(P)+1} \mid (P,o) \sqsubseteq \beta^S\} + 1;$ $\mathsf{Move}((P,o+1),(P,o+1),b);\ \mathsf{Move}(P,(P,o+1),-1)$ |
| SPLIT$(P,o,b)$ | | $\mathsf{Move}(P,(P,o,b),1);\ \mathsf{Move}((P,o+1),(P,o+1),-b)$ |

Figure 11: Main transformation primitives

tions: $\mathbf{1}_k$ denotes the vector filled with zeros but element $k$ set to 1, i.e., $(0,\ldots,0,1,0,\ldots,0)$; likewise, $\mathbf{1}_{i,j}$ denotes the matrix filled with zeros but element $(i,j)$ set to 1.

Each primitive may have a few *prerequisites*; they capture the minimal requirements on the primitive's parameters to preserve the structure of the polyhedral representation, including the invariants.

UNIMODULAR implements any per-statement unimodular transformation, extended to arbitrary iteration domains and loop nesting. U and V denote unimodular matrices. The prerequisites enforce that both matrices are $d^S$-dimensional and unimodular.

SHIFT is a kind of source-level hierarchical software pipeline, extended with parametric forward/backward iteration shifts, e.g., to delay a statement by $N$ iterations of one surrounding loop. Matrix M implements the parameterized shift of the affine schedule of a statement. M must have the same dimension as $\Gamma$.

PAD implements array padding. Matrix M allows fine-grain updates of each dimension of the array, adding an arbitrary affine expression of the global parameters. M must have the same dimension as $\Sigma_{\mathsf{A}}$

CUTDOMAIN constrains the domain with an additional inequality, given in the form of a vector $c$ with the same dimension as a row of matrix $\Gamma$.

INSERT and DELETE are the simplest statement creation and removal primitives. These primitives enforce representation invariants through careful allocation or deallocation of free slots in statement ordering vectors, thanks to the Move constructor. INSERT requires that the new statement ordering vector immediately precedes or follows an existing prefix and $\ell$ denotes the depth of the insertion.

EXTEND inserts a new intermediate loop level at depth $\ell$, initially restricted to a single iteration.

ADDLOCALVAR creates a new local variable, initially not used. This local variable is typically used in following CUTDOMAIN primitives.

PRIVATIZE and CONTRACT implement basic forms of array privatization and contraction, respectively, considering dimension $\ell$ of the array. Privatization needs an additional parameter $s$, the size of the additional dimension; $s$ is required to update the array declaration and cannot be inferred in general, because some array references may not be affine. These primitives are simple examples updating the data layout and array access functions.

FUSE and SPLIT best illustrate the benefit of designing loop transformations at the abstract semantical level of our unified polyhedral representation. First of all, loop bounds are not an issue since the code generator will handle any overlapping of iteration domains. Next, these primitives do not directly operate on loops, but consider prefixes $P$ of statement ordering vectors; they have no prerequisite and may virtually be composed with any possible transformation. For the split primitive, vector $(P,o)$ prefixes all statements concerned by the split; and parameter $b$ indicates the position where statement delaying should occur. For the fusion primitive, vector $(P,o+1)$ prefixes all statements that should be interleaved with statements prefixed by $(P,o)$. Eventually, notice that fusion followed by split (with the appropriate value of $b$) leaves the SCoP unchanged.

Although this table is not complete, it demonstrates the expressivity of the unified representation through classical *control and data* transformations.

In addition to the prerequisites in Figure 11, we have specified a number of optional *validity prerequisites* that check for the semantical soundess of the transformation, e.g., there are validity prerequisites to check that no dependence is violated by a unimodular or array contraction transformation. When exploring the space of possible transformation sequences, such validity prerequisites avoid loosing time on useless transformations.

**Note about data layout transformations.** Because the liveness of array identifier may escape the current SCoP, each update to array dimensions (e.g., through padding or privatization) should be made consistent with the rest of the program. Although we did not

address this issue yet, it is always possible to insert copy-in and/or copy-out code automatically [32, 7].

**Defining program transformations using primitives.** Let us for instance define two simple transformations, strip-mining and interchange, using primitives. INTERCHANGE$(S,o)$ swaps the roles of $\mathbf{i}_o$ and $\mathbf{i}_{o+1}$ in the schedule of $S$; it is a per-statement extension of the classical interchange making no assumption about the shape of the iteration domain. The formal definition of INTERCHANGE is provided in Figure 12. STRIPMINE$(S,o,k)$ —where $k$ is a *known integer* —introduces a new iterator to unroll $k$ times the schedule and iteration domain of $S$ at depth $o$. This transformation is a sequence of six primitives, see Figure 12. Finally, we can define two-dimensional tiling by composing strip-mining and interchange: TILE$(S,o,k)$ tiles the loops at depth $o$ and $o+1$ with $k \times k$ blocks, see Figure 12.

| Syntax | Prerequisites | Effect | Comments |
|---|---|---|---|
| INTERCHANGE$(S,o)$ | $o < d^S$ | $\mathrm{V} = \mathbf{I}_{d^S} - \mathbf{1}_{o,o} - \mathbf{1}_{o+1,o+1} + \mathbf{1}_{o,o+1} + \mathbf{1}_{o+1,o}$<br>$S \leftarrow$ UNIMODULAR$(S, \mathbf{I}_{d^S}, \mathrm{V})$ | swap rows<br>$o$ and $o+1$ |
| STRIPMINE$(S,o,k)$ | $o \leq d^S$<br>$\wedge\ k > 0$ | $S \leftarrow$ EXTEND$(S,o)$;<br>$S \leftarrow$ ADDLOCALVAR$(S)$;<br>$p = d^S + 1$; $u = d^S + d^S_{\mathrm{lv}} + d_{\mathrm{gp}} + 1$;<br>$S \leftarrow$ CUTDOMAIN$(S, \mathbf{1}_{o+1} - \mathbf{1}_o)$;<br>$S \leftarrow$ CUTDOMAIN$(S, \mathbf{1}_o - \mathbf{1}_{o+1} + (k-1)\mathbf{1}_u)$;<br>$S \leftarrow$ CUTDOMAIN$(S, \mathbf{1}_o - \mathbf{1}_p)$;<br>$S \leftarrow$ CUTDOMAIN$(S, \mathbf{1}_p - \mathbf{1}_o)$; | insert intermediate loop level<br>insert local variable<br>constant and local variables<br>$\mathbf{i}_o \leq \mathbf{i}_{o+1}$<br>$\mathbf{i}_{o+1} \leq \mathbf{i}_o + k - 1$<br>$k \times p \leq ii$<br>$ii \leq k \times p$ |
| TILE$(S,o,k)$ | $o < d^S$<br>$\wedge\ k > 0$ | $S \leftarrow$ STRIPMINE$(S,o,k)$;<br>$S \leftarrow$ STRIPMINE$(S,o+2,k)$;<br>$S \leftarrow$ INTERCHANGE$(S,o+1)$; | strip outer loop<br>strip inner loop<br>interchange in the middle |

Figure 12: Composition of transformation primitives

# 3 Compositions of Transformations

When composing long sequences of program transformations, the code ultimately generated by our framework will not necessarily be simpler than the code obtained after the same sequence of syntactic program transformations. The main asset of our framework is to completely hide the code complexity induced by long sequences of program transformations until the very end, i.e., the code generation. At intermediate steps, the complexity of the code representation within our framework remains fairly low, i.e., it only depends on the number of statements with a matrix triplet per statement; using syntactic program transformations, the code complexity may increase at each intermediate step, sometimes even preventing further optimizations as illustrated in Section 2.

### 3.1 Controling Code Complexity

With a simple example, let us compare how the code complexity may evolve at intermediate steps within a sequence of program transformations using our framework and using syntactic program transformations.

**Composing transformations without code size explosion.** The example in Figure 13 performs two matrix-vector multiplications, yielding $D =^t BEC$ (typical of quadratic form computations), where arrays B and E store $M \times N$ rectangular matrices. The left-hand side of this figure displays the code in a typical syntactic transformation framework; the right-hand side displays our polyhedral representation.

We apply a sequence of three transformations to this program The result of the first transformation is shown in Figure 14; the loops in the first nest are interchanged to optimize spatial locality of array B.[6]. The second transformation is shown in Figure 15; it fuses the two nests to improve temporal locality on array A. The result of the third transformation is shown in Figure 16; it advances the assignments to A by 4 iterations to increase instruction-level parallelism and cover the latency of floating-point multiplications.

In our framework, this sequence of transformations corresponds to the following primitives:

$$\textsc{Interchange}\big(S_1,1\big);\ \textsc{Fuse}\big((),0\big);\ \textsc{Shift}\big(S_1,\begin{bmatrix}0&0&0\\0&0&-4\end{bmatrix}\big).$$

and as shown in Figure 16, it does not complicate the program intermediate representation.

Based on the final polyhedral representation, the code generation phase will generate control-optimized code quite similar to the hand-optimized version (both shown in Figure 16), and without sacrificing efficiency in redundant guards or lost iterations.

```
    do i = 1, M
    |  do j = 1, N
(S1)|  |  A(i) += B(i,j)*C(j)
    do k = 1, M
    |  do l = 1, N
(S2)|  |  D(k) += E(l,k)*A(l)
```

$$\mathcal{D}^{S_1} = \{i,j \mid 1 \le i \le M\ ;\ 1 \le j \le N\}$$
$$\mathrm{A}^{S_1} = \begin{bmatrix}1&0\\0&1\end{bmatrix}, \Gamma^{S_1} = \begin{bmatrix}0&0&0\\0&0&0\end{bmatrix}, \beta^{S_1} = \begin{bmatrix}0\\0\\0\end{bmatrix}$$
$$\mathcal{L}^{S_1} = \big(\mathsf{A}, \begin{bmatrix}1&0&0&0&0\end{bmatrix}\big)$$
$$\mathcal{D}^{S_2} = \{k,l \mid 1 \le k \le M\ ;\ 1 \le l \le N\}$$
$$\mathrm{A}^{S_2} = \begin{bmatrix}1&0\\0&1\end{bmatrix}, \Gamma^{S_2} = \begin{bmatrix}0&0&0\\0&0&0\end{bmatrix}, \beta^{S_2} = \begin{bmatrix}1\\0\\0\end{bmatrix}$$
$$\mathcal{L}^{S_2} = \big(\mathsf{B}, \begin{bmatrix}1&0&0&0&0\end{bmatrix}\big)$$

Figure 13: Original code and its representation

[6] E is already traversed in column-major order

```
do j = 1, N
| do i = 1, M
| | A(i) += B(i,j)*C(j)
do k = 1, M
| do l = 1, N
| | D(k) += E(l,k)*A(l)
```

$$\mathcal{D}^{S_1} = \{i, j \mid 1 \leq i \leq M \; ; \; 1 \leq j \leq N\}$$
$$\mathrm{A}^{S_1} = \begin{bmatrix} \mathbf{0\,1} \\ \mathbf{1\,0} \end{bmatrix}, \Gamma^{S_1} = \begin{bmatrix} 0\,0\,0 \\ 0\,0\,0 \end{bmatrix}, \beta^{S_1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$
$$\mathcal{L}^{S_1} = \left(\mathtt{A}, \begin{bmatrix} 1\,0\,0\,0\,0 \end{bmatrix}\right)$$
$$\mathcal{D}^{S_2} = \{k, l \mid 1 \leq k \leq M \; ; \; 1 \leq l \leq N\}$$
$$\mathrm{A}^{S_2} = \begin{bmatrix} 1\,0 \\ 0\,1 \end{bmatrix}, \Gamma^{S_2} = \begin{bmatrix} 0\,0\,0 \\ 0\,0\,0 \end{bmatrix}, \beta^{S_2} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$
$$\mathcal{L}^{S_2} = \left(\mathtt{B}, \begin{bmatrix} 1\,0\,0\,0\,0 \end{bmatrix}\right)$$

Figure 14: After interchange

```
MN = min(M,N)
do x = 1, MN
| do y = 1, MN
| | A(y) += B(y,x)*C(x)
| | D(x) += E(y,x)*A(y)
| do y = MN+1, M
| | A(y) += B(y,x)*C(x)
| | D(y) += E(y,x)*A(y)
do x = MN+1, N
| do y = 1, MN
| | A(y) += B(y,x)*C(x)
| | D(y) += E(y,x)*A(y)
```

$$\mathcal{D}^{S_1} = \{i, j \mid 1 \leq i \leq M \; ; \; 1 \leq j \leq N\}$$
$$\mathrm{A}^{S_1} = \begin{bmatrix} 0\,1 \\ 1\,0 \end{bmatrix}, \Gamma^{S_1} = \begin{bmatrix} 0\,0\,0 \\ 0\,0\,0 \end{bmatrix}, \beta^{S_1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$
$$\mathcal{L}^{S_1} = \left(\mathtt{A}, \begin{bmatrix} 1\,0\,0\,0\,0 \end{bmatrix}\right)$$
$$\mathcal{D}^{S_2} = \{k, l \mid 1 \leq k \leq M \; ; \; 1 \leq l \leq N\}$$
$$\mathrm{A}^{S_2} = \begin{bmatrix} 1\,0 \\ 0\,1 \end{bmatrix}, \Gamma^{S_2} = \begin{bmatrix} 0\,0\,0 \\ 0\,0\,0 \end{bmatrix}, \beta^{S_2} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \end{bmatrix}$$
$$\mathcal{L}^{S_2} = \left(\mathtt{B}, \begin{bmatrix} 1\,0\,0\,0\,0 \end{bmatrix}\right)$$

Figure 15: After fusion

**More about code size.** With syntactic program transformations, code complexity is due —in a large part —to control optimizations (hoisting of conditionals, unrolling) which do not affect the complexity of our representation. This is a major difference with the simpler polyhedral representations used in HPF compilers (to implement data distribution and generate communications) or MARS [23], where domain and schedule information are mixed together. When applying transformation sequences in our framework, schedule and domain updates are independent and the final imperative code generation phase takes care of producing a control-efficient loop nest; see Section 4.1. For instance, loop unrolling is done in a *lazy* fashion: we perform strip-mining instead, delaying the proper unrolling to the code generation phase. Separate transformations of the virtually unrolled iterations are still possible.

```
MN = min(M-4,N)
do x = 1, MN
| A(1) += B(1,x)*C(x)
| A(2) += B(2,x)*C(x)
| A(3) += B(3,x)*C(x)
| A(4) += B(4,x)*C(x)
| do y = 1, MN
| | A(y+4) += B(y+4,x)*C(x)
| | D(x) += E(y,x)*A(y)
| D(x) += E(MN-3,x)*A(MN-3)
| D(x) += E(MN-2,x)*A(MN-2)
| D(x) += E(MN-1,x)*A(MN-1)
| D(x) += E(MN,x)*A(MN)
| do y = MN+1, M-4
| | A(y+4) += B(y+4,x)*C(x)
| | D(y) += E(y,x)*A(y)
| D(M-3) += E(MN-3,x)*A(MN-3)
| D(M-2) += E(MN-2,x)*A(MN-2)
| D(M-1) += E(MN-1,x)*A(MN-1)
| D(M) += E(MN,x)*A(MN)
do x = MN+1, N
| A(1) += B(1,x)*C(x)
| A(2) += B(2,x)*C(x)
| A(3) += B(3,x)*C(x)
| A(4) += B(4,x)*C(x)
| do y = 1, M-4
| | A(y+4) += B(y+4,x)*C(x)
| | D(y) += E(y,x)*A(y)
| D(M-3) += E(MN-3,M-3)*A(MN-3)
| D(M-2) += E(MN-2,M-2)*A(MN-2)
| D(M-1) += E(MN-1,M-1)*A(MN-1)
| D(M) += E(MN,M)*A(MN)
```

$$\mathcal{D}^{S_1} = \{i, j \mid 1 \leq i \leq M \; ; \; 1 \leq j \leq N\}$$
$$\mathrm{A}^{S_1} = \begin{bmatrix} 0\,1 \\ 1\,0 \end{bmatrix}, \Gamma^{S_1} = \begin{bmatrix} \mathbf{0\,0\;\;0} \\ \mathbf{0\,0\;\text{-}4} \end{bmatrix}, \beta^{S_1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$
$$\mathcal{L}^{S_1} = \left(\mathsf{A}, \begin{bmatrix} 1\,0\,0\,0\,0 \end{bmatrix}\right)$$
$$\mathcal{D}^{S_2} = \{k, l \mid 1 \leq k \leq M \; ; \; 1 \leq l \leq N\}$$
$$\mathrm{A}^{S_2} = \begin{bmatrix} 1\,0 \\ 0\,1 \end{bmatrix}, \Gamma^{S_2} = \begin{bmatrix} 0\,0\,0 \\ 0\,0\,0 \end{bmatrix}, \beta^{S_2} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$
$$\mathcal{L}^{S_2} = \left(\mathsf{B}, \begin{bmatrix} 1\,0\,0\,0\,0 \end{bmatrix}\right)$$

Figure 16: After four shifts

**Search space properties.** Confluence and commutativity properties are additional benefits of the separation principle. Of course, data and control transformations commute, but surprisingly, inter-statement and intra-statement iteration reordering as well, e.g., loop fusion commutes with loop interchange. Confluence properties are also available but their structure is still not well understood. Such properties are useful in the context of iterative searches; they may dramatically reduce the search space, and they also improve the understanding of its structure, which in turns enables more efficient search strategies [16].

### 3.2 Searching for Compositions

Let us now describe how iterative searches may benefit from our contributions. Using the framework two search approaches are possible: the incremental composition of program transformations, and direct search of code representation parameters.

The incremental composition of program transformations simply consists in composing sequences of adhoc program transformations (or primitives). In many cases, applying a program transformation amounts to recomputing the matrix triplets of every statement. With the incremental approach, the framework fulfills its primary role: enabling the easy composition of arbitrarily long sequences of program transformations without increasing code complexity. Only the matrix triplets of each statement are modified after each transformation, as illustrated in the previous section, while the code complexity is incrementally increased using syntactic transformations. Only transformations which increase the number of statements can increase the complexity of the representation, such as prefetching; and still, the complexity increase remains fairly moderate in most cases.

The code representation framework also opens up a new approach for searching compositions of program transformations. Since many program transformations have the only effect of modifying the parameters of the matrix triplets, then an alternative is to *directly search the matrix parameters themselves*. In some cases, changing one or a few parameters is equivalent to performing a sequence of program transformations, so that searching for compositions of program transformations is much more systematic and simple. For instance, let us consider again the example of Section 3.1, and let us assume the matrix parameters of A, $\Gamma$ and $\beta$ of the statements in the original representation are being systematically searched. The point in the search space corresponding to the values of the matrix parameters in Figure 16 also corresponds to a composition of the transformations $\text{INTERCHANGE}(S_1,1)$; $\text{FUSE}((),0)$; $\text{SHIFT}(S_1, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -4 \end{bmatrix})$.

This search approach needs to take into account the impact of invariants and validity prerequisites on matrix parameters, i.e., the search space. Also, this approach is still not amenable to program transformations that increase/decrease the number of statements. Future work may include adapting the model to make an even greater number of transformations amenable to this search approach.

## 4 Implementation

The whole infrastructure is implemented as a plug-in for Open64/ORC and is publicly available at `http://www-rocq.inria.fr/a3/wrap-it`.

### 4.1 Code Generation

After polyhedral transformations, the regeneration of imperative loop structures is the last step to the final program. It has a strong impact on the target code quality: we must ensure that no redundant guard or complex loop bound spoils performance gains achieved with polyhedral transformations. We used the recent Quilleré et al. method [28] with some additional improvements to guarantee the absence of duplicated control. The modified algorithm, CLooG, is presented and evaluated in [1].

The most computationally intensive task of the code generation process consists in separating the statement polyhedra into disjoint convex polyhedra. The statement polyhedra are obtained by merging the iteration domain and schedule polyhedra of each statement in a single polyhedron. In terms of polyhedral operations, the worst-case complexity is $O(3^n)$. We will investigate methods for further reducing this complexity. The CLooG generator can handle all 12 benchmarks in Figure 17. Experiments were conducted on a $512\,MB$ $1\,GHz$ Pentium III machine; generation times range from 1 to 127 seconds (34 seconds on average).

### 4.2 WRaP-IT: WHIRL Represented as Polyhedra Interface Tool

To apply our framework to full benchmarks, the first requirement was to streamline the extraction of static control parts and code generation, achieving a good integration of polyhedral techniques into optimizing and parallelizing compilers. This interface tool is built on top of Open64. It converts the WHIRL —the compiler's hierarchical intermediate representation —to an augmented polyhedral representation, maintaining a correspondence between matrices in SCoP descriptions with the symbol table and syntax tree. This representation is called the WRaP: *WHIRL Represented as Polyhedra*. It is the basis for any polyhedral analysis or transformation. Then, the second part of the tool is a modified version of the CLooG polyhedral code generator [1]; it regenerates a WHIRL syntax tree from the WRaP. The whole interface tool is called WRaP-IT. WRaP-IT may be used in a normal compilation flow as well as in a source-to-source framework; more details can be found in [2]. Although WRaP-IT is still a prototype, it proved to be very robust; the whole source-to-polyhedra-to-source transformation was successfully applied to all 12 benchmarks in Figure 17.

### 4.3 Extracting SCoPs

SCoP extraction is greatly simplified when implemented within a modern compiler infrastructure such as ORC. Previous phases include function inlining, constant propagation, loop normalization, integer comparison normalization, dead-code and `goto` elimination, and in-

duction variable substitution, along with language-specific preprocessing: pointer arithmetic is replaced by arrays, pointer analysis information is available, etc.

| | SCoPs | | Statements | | Array References | |
|---|---|---|---|---|---|---|
| | All | Parametric | All | in SCoPs | All | Affine |
| applu | 19 | 15 | 757 | 633 | 1245 | 100% |
| apsi | 80 | 80 | 2192 | 1839 | 977 | 78% |
| art | 28 | 27 | 499 | 343 | 52 | 100% |
| lucas | 4 | 4 | 2070 | 2050 | 411 | 40% |
| mgrid | 12 | 12 | 369 | 369 | 176 | 99% |
| quake | 20 | 14 | 639 | 489 | 218 | 100% |
| swim | 6 | 6 | 123 | 123 | 192 | 100% |
| adm | 80 | 80 | 2260 | 1899 | 147 | 95% |
| dyfesm | 75 | 70 | 1497 | 1280 | 507 | 99% |
| mdg | 17 | 17 | 530 | 464 | 355 | 84% |
| mg3d | 39 | 39 | 1442 | 1242 | 1274 | 19% |
| qcd | 30 | 23 | 819 | 641 | 943 | 100% |

Figure 17: Static control parts in high-performance applications

Figure 17 summarizes the results for the SpecFP 2000 and PerfectClub benchmarks handled by our tool (single-file programs only at the time being, without interprocedural analysis or inlining). Construction of the unified polyhedral representation takes much less time than the preliminary analyses performed by Open64. All codes are in Fortran77, except art and quake in C, and lucas in Fortran90. The first two columns count the number of SCoPs and those holding at least one global parameter. The next two columns in the "Statements" section demonstrate the good covering of program statements by SCoPs (many statements are enclosed in affine loops). The last two columns in the "Array References" section are very promising for dependence analysis: most subscripts are affine except for lucas and mg3d; moreover, the rate is over 99% in 7 benchmarks, but approximate array dependence analyses will be required for a good coverage of the 5 others. In accordance with earlier results using Polaris [8], the coverage of regular loop nests is strongly influenced by the quality of the constant propagation, loop normalization and induction variable detection.

### 4.4 URUK: Unified Representation Universal Kernel

URUK is the key software component for our framework: it applies compositions of primitives to the WRaP representation. A scripting language defines transformations and enables

the composition of new transformations. Each transformation is defined by its name, parameters, prerequisites, and its effect. Nested calls of transformations are allowed.

The definition is written in C++ with overloaded operators for vector and matrices, such as `<<` for the lexicographic ordering $\ll$ and `<=` for the prefix order $\sqsubseteq$. Figure 18 shows the definition of the Move constructor we defined in Section 2.4, and Figure 19 defines the SPLIT transformation primitive calling Move itself. From those definitions, URUK produces the source code of the related transformations; it effectively generates a class for each transformation with its own methods for prerequisite checking and application.

```
%transformation move

%param BetaPrefix P, Q
%param Offset o

%prereq P<=Q

%code
{ foreach S in SCoP
  { if      ((P<=S.Beta)&&(Q<=S.Beta)) S.Beta(P.dim())+=o;
    else if ((P<=S.Beta)&&(Q<<S.Beta)) S.Beta(P.dim())+=o; }}
```

Figure 18: Definition of constructor Move

```
%transformation split

%param BetaPrefix P
%param Offset o, b

%code
{ UrukVector Q=P; Q.enqueue(o); Q.enqueue(b);
  UrukVector R=P; R.enqueue(o+1);
  UT_move(P,Q,1).apply(SCoP);
  UT_move(R,R,-1).apply(SCoP); }
```

Figure 19: Split primitive definition

# 5 Related Work

Loop restructuring compilers introduced several unified models and intermediate representations for loop transformations, but none of them addressed the general composition and parameterization problem of polyhedral techniques. ParaScope [6] is both a dependence-based framework and an interactive source-to-source compiler for Fortran; it implements classical loop transformations. SUIF [14] was designed as an intermediate representation and framework for complex program transformations; it quickly became a standard platform

for implementing virtually any optimization prototype. Polaris [3] is a parallelizing compiler for Fortran; it features a rich sequence of analyzes and loop transformations applicable to real benchmarks. These three projects are based on a syntax-tree representation, ad-hoc dependence models and implement polynomial algorithms. PIPS [15] is probably the most complete loop restructuring compiler, implementing polyhedral analyses and transformations (including affine scheduling) and interprocedural analyses (array regions, alias). PIPS uses an expressive syntax tree representation with polyhedral annotations.

Two codesign tools share a lot of motivations and technology with our semi-automatic optimization project. MMAlpha [13] is a domain-specific single assignment language for systolic array computations, a polyhedral transformation framework, and a high-level circuit synthesis tool. The interactive and semi-automatic approach to polyhedral transformations were introduced by MMAlpha. The PICO project [30] is a more pragmatic approach to codesign, restricting the application domain to loop nests with uniform dependences and aiming at the selection and coordination of existing functional units to generate an application-specific VLIW processor. Both tools only target small kernels.

Within the Omega project [18], the Petit dependence analyzer and loop restructuring tool [17] is closer to our work: it provides a unified polyhedral framework (space-time mappings) for iteration reordering only, and it shares our emphasis on per-statement transformations. It is intended as a research tool for small kernels only.

Most iterative optimization works use classical optimizing compiler frameworks with syntactic intermediate representations. Recently, the MARS compiler [23] has been applied to iterative optimization [24]. This compiler is based on a polyhedral representation, aiming at the unification of classical dependence-based loop transformations with data storage optimizations. Its successes in iterative optimization [24] makes it the main comparison point for our work. However, the MARS intermediate representation captures only part of the loop-specific information, namely the iteration domain and array access functions; it lacks the characterization of *iteration orderdings* through *multidimensional affine schedules* [11].

# 6 Conclusion

We presented a polyhedral framework that enables the composition of long sequences of program transformations. Coupled with a robust code generator, this method avoids the typical code complexity explosion of long compositions of program transformations; these techniques have been implemented in the Open64/ORC compiler. The ability to perform numerous compositions of program transformations is key to the extension of iterative optimizations to finding the appropriate program transformations instead of just the appropriate program transformation parameters. We have also shown that, in many cases, searching

for compositions of transformations is amenable to searching for parameters of the matrix triplets of each individual statement, opening even faster and more efficient searching techniques. Next, we will investigate the properties of the new search space defined by this framework.

# References


[1] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'2 IEEE International Symposium on Parallel and Distributed Computing*, Ljubjana, Slovenia, Oct. 2003.

[2] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, LNCS, College Station, Texas, Oct. 2003.

[3] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

[4] J.-F. Collard. Automatic parallelization of while-loops using speculative execution. *Int. J. of Parallel Programming*, 23(2):191–219, Apr. 1995.

[5] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *ACM Symp. on Principles and Practice of Parallel Programming*, pages 92–102, Santa Barbara, California, July 1995.

[6] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, 1993.

[7] B. Creusillet. *Array Region Analyses and Applications*. PhD thesis, École Nationale Supérieure des Mines de Paris (ENSMP), Paris, France, Dec. 1996.

[8] R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the perfect benchmarks. *IEEE Trans. on Parallel and Distributed Systems*, 9(1):5–23, Jan. 1998.

[9] P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, St. Malo, France, July 1988.

[10] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, Sept. 1988.

[11] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, Dec. 1992. See also Part I, one dimensional time, 21(5):315–348.

[12] G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *11$^{th}$ Workshop on Languages and Compilers for Parallel Computing*, LNCS, Washington DC, July 2002. Springer-Verlag.

[13] A.-C. Guillou, F. Quilleré, P. Quinton, S. Rajopadhye, and T. Risset. Hardware design methodology with the Alpha language. In *FDL'01*, Lyon, France, Sept. 2001.

[14] M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.

[15] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *ACM Int. Conf. on Supercomputing (ICS'2)*, Cologne, Germany, June 1991.

[16] D. S. K. D. Cooper and and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. of Supercomputing*, 2002.

[17] W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, University of Maryland, 1996.

[18] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symp. on the frontiers of massively parallel computation*, McLean, 1995.

[19] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *Proc. CPC'10 (Compilers for Parallel Computers)*, pages 35–44, 2000.

[20] A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *24$^{th}$ ACM Symp. on Principles of Programming Languages*, pages 201–214, Paris, France, jan 1997.

[21] V. Loechner and D. Wilde. Parameterized polyhedra and their vertices. *Int. J. of Parallel Programming*, 25(6), Dec. 1997. `http://icps.u-strasbg.fr/PolyLib`.

[22] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *ACM Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, 6, pages 94–104, 1996.

[23] M. O'Boyle. MARS: a distributed memory approach to shared memory compilation. In *Proc. Language, Compilers and Runtime Systems for Scalable Computing*, Pittsburgh, May 1998. Springer-Verlag.

[24] M. O'Boyle, P. Knijnenburg, and G. Fursin. Feedback assisted iterative compiplation. In *Parallel Architectures and Compilation Techniques (PACT'01)*. IEEE Computer Society Press, Oct. 2001.

[25] Open research compiler. `http://ipf-orc.sourceforge.net`.

[26] D. Parello, O. Temam, and J.-M. Verdun. On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance? matrix-multiply revisited. In *SuperComputing'02*, Baltimore, Maryland, Nov. 2002.

[27] G.-R. Perrin and A. Darte, editors. *The Data Parallel Programming Model*. Number 1132 in LNCS. Springer-Verlag, 1996.

[28] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. J. of Parallel Programming*, 28(5):469–498, october 2000.

[29] L. Rauchwerger and D. Padua. The LRPD test: Speculative run–time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems, Special Issue on Compilers and Languages for Parallel and Distributed Computers*, 10(2):160–180, 1999.

[30] R. Schreiber, S. Aditya, B. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of nonprogrammable hardware accelerators. Technical report, Hewlett-Packard, May 2000.

[31] M. M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independant storage mapping for loops. In *ACM Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, 8, 1998.

[32] P. Tu and D. Padua. Automatic array privatization. In *6$^{th}$ Workshop on Languages and Compilers for Parallel Computing*, number 768 in LNCS, pages 500–521, Portland, Oregon, Aug. 1993.

[33] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[34] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[35] D. G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, University of Maryland, 1995.

[36] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *ACM Symp. on Programming Language Design and Implementation (PLDI'03)*, San Diego, California, June 2003.

Unité de recherche INRIA Futurs
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifi que
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399